PORTAL
THE ACM DIGITAL LIBRARY

Try the *new* Portal design

Give us your opinion after using it.

Citation

Symposium on Applied Computing >archive
Proceedings of the 2000 ACM symposium on Applied computing >toc
2000 , Como, Italy

A tool for Internet-oriented knowledge based systems

Author
Robert Inder

Sponsor
SIGAPP : ACM Special Interest Group on Applied Computing

http://doi.acm.org/10.1145/335603.335625 (Use this link to Bookmark this page)

> full text   > references   > index terms   > peer to peer

> Discuss   > Similar   > Review this Article   Save to Binder

> BibTex Format

↑ **FULL TEXT:**   Access Rules

pdf 562 KB

↑ **REFERENCES**

Note: OCR errors may be found in this Reference List extracted from the full text article. ACM has opted to expose the complete List rather than only correct and linked references.

1   A. Sloman and S. Hardy. "Poplog: A Multi-Purpose Multi-Language Program Development Environment" AISB Quarterly, Vol 47, pp26-34, t983.

2   Gary Riley, "Clips: An Expert System Building Tool". Proceedings of the Techlology 2001 Conference, San Jose, CA, 1991

3   Larry Wall , Tom Christiansen , Randal L. Schwartz , Stephan Potter, Programming Perl (2nd ed.), O'Reilly & Associates, Inc., Sebastopol, CA, 1996

4   Joseph C. Giarratano , Gary Riley, Expert Systems: Principles and Programming, PWS Publishing Co., Boston, MA, 1994

5   S. Srinivasan, "Advanced Perl Programming". O'Reilly and Associates, 1997.

6   R. Inder, "State of the ART: A review of the Automated Reasoning Tool." In Expert System Applications, S. Vadera (ed), Sigma Press, Wilmslow, 1989. Also available as AIAI-TR-41.

7   Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence 19(1982), 17-37.

8   Robert inder, Nadia Bianchi and Toshikazu Kato, "K- DIME: A Software Framework for Kansei Filtering of [nternet Material" Proceedings of IEEE Systems Man and Cybernetics, 1999~

9   Robert [nder and Toshikazu Kato, "Towards Shoppers' Assistants: Agents to Combine Information". In Advanced Database Systems/or Integration of Media and User Environments 98, Y~ Kambayashi (ed.), World Scientific, February 1998.

10   Robert Inder, Matthew Hurst and Toshkazu Kato, "A Prototype Agent to Assist Shoppers" in Computer Networks and ISDN Systems, V30 (1998), pp 643-645. Full version available as Technical Report ETL*TR98. 3, Electrotechnical Laboratory, Tsukuba, Japan.

11   Kozo Sugiyama and Kazuo Misue, "Visualization of Structural information: Automatic Drawing of Compound Digraphs,". IEEE Transactions on Systems, Man, and Cybernetics, vol. 21. 1991.

12   Robert Indef. CAe~ Users Manual. Technical Report ETL-TR98-3,~ Electrotechnical Laboratory, Tsukuba, Japan.

⇑ INDEX TERMS

Primary Classification:
 H. Information Systems
  ↳ **H.4** INFORMATION SYSTEMS APPLICATIONS
   ↳ **H.4.3** Communications Applications

    ↳ **Nouns:** Internet

Additional Classification:
 D. Software
  ↳ **D.4** OPERATING SYSTEMS
   ↳ **D.4.9** Systems Programs and Utilities

    ↳ **Nouns:** perl

 I. Computing Methodologies
  ↳ **I.2** ARTIFICIAL INTELLIGENCE

General Terms:
Design, Languages, Management, Performance, Theory

Keywords:
CLIPS, Perl, knowledge-based system tools, rule-based programming

↑ **Peer to Peer - Readers of this Article have also read:**

❖ Data structures for quadtree approximation and compression
Communications of the ACM 28, 9
Hanan Samet

❖ 3D representations for software visualization
Proceedings of the 2003 ACM symposium on Software visualization
Andrian Marcus , Louis Feng , Jonathan I. Maletic

❖ Boundary and Object Detection in Real World Images
Journal of the ACM (JACM) 23, 4
Yoram Yakimovsky

❖ The state of the art in automating usability evaluation of user interfaces
ACM Computing Surveys (CSUR) 33, 4
Melody Y. Ivory , Marti A Hearst

❖ Image-based objects
Proceedings of the 1999 symposium on Interactive 3D graphics
Manuel M. Oliveira , Gary Bishop

A Tool for Internet-Oriented Knowledge Based Systems

Robert Inder
Division of Informatics
University of Edinburgh
Edinburgh, Scotland
R.Inder@ed.ac.uk

ABSTRACT

This paper describes CAPE, a programming environment
that combines Clips And Perl with Extensions. CLIPS is
an efficient and expressive forward-chaining rule-based sys-
tem with a flexible object system (supporting both message
passing and generic functions). Perl is a popular proce-
dural language with extremely powerful regular expression
matching facilities, and a huge library of freely available
software modules. CAPE closely integrates these two pro-
gramming languages, and provides extensions to facilitate
building systems with an intimate mixture of the two lan-
guages. These features make CAPE an excellent language for
building knowledge-based systems to exploit the opportuni-
ties being presented by the Internet.

This paper describes the current version of CAPE and the
facilities it offers programmers, including the demonstration
systems and "component applications" that are distributed
with it. The use of the system is then discussed with refer-
ence to an application for automatically generating graphs
of remote web sites. Finally, planned developments of the
system are indicated.

General Terms

Rule-based Programming, Knowledge-Based System Tools,
CLIPS, Perl

1. BACKGROUND

Conventional Knowledge Based Systems (KBSs) involve the
controlled manipulation of symbolic descriptions of the world.
Extracting such symbolic descriptions from sensory data was
(and still is) a major challenge in its own right, addressed
by specific fields such as speech recognition and vision, and
most work in KBSs tries to sidestep it.

At first, KBSs worked with very restricted amounts of in-
formation (e.g. chess positions) and typically solving prob-
lems described in special format data files. They began to
achieve much wider use/acceptance when Expert Systems
began to ask questions of users, ("Has the patient got a

rash?"), thereby using the user's perceptual abilities and
common sense to analyse the world into the abstract terms
that the KBSs required. Subsequently, KBSs began to access
data, either statically (i.e. KBS linked to database) or by
analysing real-time feeds (e.g. network monitoring). To a
greater or lesser extent, these systems can determine what
data is relevant, and when, but only within a fixed (or at
least limited) range of data for which the format and seman-
tics were known to the system builders (or maintainers).

Now, we have the Internet.

By providing "universal" connectivity between computers,
the Internet has created a previously inconceivable range of
possibilities for any program. Innumerable documents —
ranging from the ephemera of electronic "news" discussions
through to published documents (including newspapers, lit-
erature and technical documents) — can be retrieved from
almost anywhere in the world within moments. Programs
can instantly query a multitude of databases, covering ev-
erything from holiday prices to registered trade marks, and
call on other computational services such as dynamic infor-
mation feeds (e.g. stock prices), notification of changes and
so forth. Finally, the Internet allows programs to easily in-
teract with huge numbers of people, either by email or using
Web-based forms.

The number of sources of information and services continues
to grow at an incomprehensible pace. Crucially, though,
those sources are outside the control of—indeed, since they
are continually growing and evolving, outside the *knowledge*
of—the system's builders.

Fully exploiting the opportunity presented by the Internet—
e.g. by building systems which can analyse and filter infor-
mation for a particular purpose—will in several ways require
a new breed of KBSs. They will, at the very least, have to be
easily configured to deal with new sources of information,
and should, more importantly, be able to explore, discover
and operate within a huge and ever-changing world of infor-
mation.

By making it easier to both locate and distribute software,
the rise of the Internet has also greatly increased the num-
ber that readily accessible software packages. These include
not only end-user applications, "plugins" and extensions for
system tools and programming languages, but also library
components for more-or-less standard tasks, such as parsing
standard document formats and mark-ups.

In particular, there are efficient implementations of data-
intensive algorithms such as neural nets, statistical analy-
sis or "data mining" tools, and information retrieval and
other techniques for processing free text based on word oc-

currence statistics. Such packages are potentially valuable new sources of information. It will therefore be increasingly important that software tools are able to work with them—either by incorporating libraries or by interfacing to stand-alone packages.

2. REQUIREMENTS
To best exploit the opportunities presented by the rise of Internet, a KBS programming tool should offer ...

Expressive language The language should combine object oriented programming and symbolic reasoning. It should also provide efficient memory management, and support both incremental program development and interactive debugging.

Searching and pattern matching Only a very small proportion of the oceans of material available can be readily assimilated by a program. Most is free text, and much of the rest—such as results of database queries—is tables and other more-or-less regular "arrays". Handling such material requires searching for words and phrases, recognising repeating structures and extracting content from them.

Support for Server Building The Internet makes it natural to make a system available as a "server" that users—or other programs (i.e. agents)—can contact as required. However, not all KBSS are well suited to starting on demand (e.g. for CGI), and many can require long periods of processing. An ideal KBS tool should support building processes that remain responsive while reasoning.

Easy Interaction with Other software As well as providing services to other programs, future KBS must be able to use the wide range of libraries or packages that are available. Tools must allow linking to C (the most common language for package distribution), and facilitate interacting with other stand-along processes.

One way to achieve this is by building libraries for a language which has some of the desired properties. This is the approach taken by Jess, the Java Expert System Shell (see http://herzberg1.ca.sandia.gov/jess/), which implements a rule-based engine within Java.

The alternative is to combine two (or more) languages which each exhibit some of the desired properties into a single programming environment, in the manner of Poplog [1]. This is the approach adopted by CAPE.

3. CAPE
CAPE (Clips And Perl with Extensions) is a programming environment which allows programs to be written in an intimate mixture of the CLIPS [2] (CLIPS: C Language Integrated Production System) rule-based and object-oriented language, and Perl [3], a procedural programming language. CLIPS was chosen because it closely integrates a very fast forward chaining rule-based system with a flexible object system that supports both message passing and generic functions. CLIPS was initially a partial re-implementation, in C, of Inference Art [6], which was arguably the most powerful of the lisp-based "knowledge representation tookits" that emerged during the mid/late 1980s. Its rule language features very powerful and efficient pattern matching facilities

based on the RETE algorithm [7], and including the ability to match against the state of objects, and a truth maintenance mechanism. There is tutorial material for CLIPS in [4], and CLIPS itself is accessible via

http://www.ghgcorp.com/clips/

along with detailed manuals and pointers to related software and information.

Perl was chosen because of its extremely powerful regular expression matching facilities, and its huge library of freely available software modules. It also supports complex data structures, and (combinations of symbolic patterns), There are many books about Perl programming (e.g. [5]). Perl itself is accessible via, http://www.perl.com/, along with manuals and pointers to a huge quantity of related software and information.

To the two separate language sub-systems, CAPE adds inter-calling and data transfer. It also provides mechanisms for synchronising the initialisation of data structures between the two programming systems. Finally, it uses CLIPS's *run function* facility to support monitoring Internet sockets while reasoning, allowing the system to remain responsive to socket activity while reasoning.

3.1 CAPE Program Structure
When CAPE reads a program, it starts by breaking the stream of characters read into "chunks" of either CLIPS or Perl code, or CAPE's own commands. Because CLIPS's syntax is extremely simple, valid CLIPS code can be recognised from the first non-blank character. Since CAPE's own commands are required to be prefixed by an "!", anything else is treated as Perl code. The nature of the chunk being read is then used to determine how it will end. Balancing parentheses makes this easy for CLIPS code. Perl chunks are terminated by heuristics based on commonly followed layout conventions, but CAPE also provides mechanisms to forcibly terminate a chunk if these heuristics are inappropriate.

Once a chunk has been read, it is pre-processed. CAPE allows textual substitutions to be defined using Perl regular expressions, and for arbitrary user-defined functions to be called. CAPE itself as yet makes only minimal use of this mechanism (to allow Perl scalar variables to be accessed from within CLIPS code), but the mechanism is provided primarily as a hook to support future language extensions.

Finally, the pre-processed code is passed to one or other of the two language interpreters for execution.

Finally, CAPE offers the user a read/eval/print loop, which prompts for a command, reads and pre-processes a chunk as described above, sends it for evaluation by either the Perl or Clips interpreter, and ensures that the result is printed. Listener itself is in Perl, so can be redefined, and thus customised or extended, by user.

3.2 Inter-Language Interaction
The most general bridge between CLIPS and Perl is through evaluation. Both languages provide an eval function which takes a string argument that it parses and executes. Both these functions are made available to the "other" language, along with a function which interprets its argument as a CAPE "chunk" which is classified as described above. String to be evaluated through this mechanism are pre-processed in the same way as code that CAPE has read.

In addition to this completely general evaluation mechanism, CAPE also provides functions for directly calling named

35

functions, and for sending messages. When these functions are called, C code within CAPE maps their arguments directly from the C data structures underlying the calling language into the corresponding C data structures for the target language. Any result returned by the called function is similarly transformed into the form required by the calling language. Since both CLIPS and Perl employ dynamic memory allocation and garbage collection, the code for mapping arguments between the systems must itself allocate (and deallocate) memory in order to avoid problems caused by the interaction of the two memory management systems.

CAPE itself only supports the mapping of simple data types (strings and numbers, and lists of these) between the component languages. This is felt to be a good compromise between power and simplicity of implementation (and thus comprehensibilty and reliability). It would be possible to provide more complex mappings (e.g. transforming a complete CLIPS object into a Perl hash or object). However, there is as yet no compelling case for providing such mappings in isolation, as opposed to as part of a comprehensive integration of the languages' object systems, which while desirable (see "Future work" below), clearly involves considerable design, programming and testing effort.

The facilities just described allow a CLIPS programs to call specific Perl subroutines, evaluate strings or match Perl regular expressions, on either the left- or right-hand side of a rule—that is, in either the condition or the action part—or from within the body of a CLIPS function or message handler. In particular, rule firing can be made dependent on successful matching of Perl regular expressions.

In the other direction, Perl programs can call functions defined in CLIPS—both normal functions and generic functions—and can send messages to CLIPS objects. There are also Perl subroutines defined for asserting facts into CLIPS working memory, and for having CLIPS evaluate an arbitrary string.

3.3 Sockets
CAPE provides functions for initiating and configuring Internet socket handling, and for monitoring sockets while the CLIPS rule engine is running. These functions are available from both CLIPS and Perl. The current state of port monitoring and socket connection is used to continually update a collection of CLIPS objects which are accessible from both languages and available for pattern matching in CLIPS rules. This means systems to provide services via an Internet port can be built using only CAPE itself.

The lowest level handling of socket activity is done in C within the core of CAPE. CLIPS is able to call an arbitrary function after each rule firing, and CAPE uses this to check for and accept connections to any ports being monitored, and to aggregate any data received from any current connection. Then, whenever the buffer associated with a connection is full, or its record break character (currently new line) is received, the accumulated byte string is passed to a connection-specific Perl function for filtering. This function could simply respond directly to the input received. Typically, though, it will map it into CLIPS working memory, thereby allowing the full power of the pattern matching to be used to decide when and how to respond.

4. PROGRAM STRUCTURE
The CLIPS rule engine has a very powerful mechanism for deciding which rule to fire at any point. This can be used to make extremely subtle and flexible decisions about the flow of control within the system—i.e. about what the CAPE application should do next. Doing this requires structuring the system as a whole as a rule-based system, so that firing rules triggers activities that are possibly implemented in Perl. A system implemented in this way can remain responsive to external events, provided that the code executed by any particular rule does not take too long to run. Perl is used to help map "the world" into symbols that CLIPS can then reason about, and then to interpret the symbolic results of that reasoning into actions in the world.

CAPE has been designed (and primarily tested and exercised) with this model of system operation in mind. However, there is no (known!) obstacle to building an "inverted" system — i.e. one in which overall control resides in a Perl program (or an interface generating call-backs into Perl), within which some subroutines specify or initiate rule-based reasoning.

5. USING CAPE
In addition to the demonstration and component applications distributed with the system (see below), CAPE is being used to develop DIME: the Distributed Information Manipulation Environment. DIME is a collection of components useful for building systems to retrieve and manipulate distributed data. These components include a highly flexible document cache (which is able to fetch documents when necessary, store them locally, and search them in various ways) and a framework for specifying and controlling interaction with remote search systems.

DIME is being used to develop two research systems. The first is K-DIME, or Kansei DIME [8], which is concerned with fetching and filtering images based on subjective criteria, or kansei. The second system being build using DIME is Maxwell, a "smart" agent intended to both query and combine the results from a number of book vendors databases on behalf of the user. An initial version of Maxwell was implemented in Emacs Lisp (see [9] and [10]), and a more powerful version is currently being implemented in CAPE using DIME. The remainder of this section will illustrate the way various features of CAPE are used and interact by describing a system for generating graphical maps of web sites based on the structure of the site and the nature of the pages it contains.

Given an initial URL and a regular expression to delimit the area to be mapped, the system fetches the relevant pages and analyses them to extract the links that they contain. The eventual aim is to generate a graph and output it in the format required by a graph layout system. The example system is geared towards the layout system found in the "thinking support tool" D-Abductor [11].

Although one can trivially generate a graph description from a web site, such approaches do not scale well: even modest web sites give rise to graphs that contain too many nodes to be drawn effectively. Producing workable graphs, therefore, involves either generating or recognising structures that can be used as a basis for omitting or temporarily hiding parts of the graph.

Different graph layout systems will hit size limits at different points. Indeed, the layout system used in this system is not particularly good in this respect, since because of the way it arranges nodes to reflect graph structure, certain site topologies cause it to make very inefficient use of

screen space. However, that simply means that the limitations of drawing complete graphs are apparent a sooner, so the scope for intelligently transforming the site description to overcome them can be explored while working with smaller web sites.

A system for mapping a remote web site must fetch the pages that are relevant to the map, identify their inter-connections, and their links to the outside world, and then decide how they can be clustered, and which links, pages, servers and clusters should be shown in the particular graph.

Our programming language should ideally:

- allow decisions to be expressed clearly in "domain" terms, such as web pages, servers, links, graph nodes and so forth

- use standard libraries for actually fetching documents

- provide good pattern matching for analysing URLs and extracting information from document contents

CAPE meets these criteria well.
The mapping system uses Perl for

Accessing libraries: Pages are fetched through the document cache component application, which in turn uses the HTTP package from CPAN.

Manipulating URLs: Extracting and canonicalising protocol, host names, file extension etc.

Extracting links: Using regular expressions to search the pages retrieved to look for links, image maps and so forth,

Gathering statistics: Using regular expressions to analyse the documents handled, finding and counting links, words in anchor texts and so forth.

However, the top-level of the system is written in CLIPS. In particular, the system's ontology is described by defining CLIPS objects for such concepts as pages, page sets, servers and directories. CLIPS rules and methods are then used for mapping pages into suitable graph descriptions. The current prototype contains just over 40 rules, doing such things as:

- recognising and collapsing multiple links between the same nodes,

- recognising and grouping (and potentially collapsing) sets of similarly-linked pages

- selecting the necessary number of strategies for omitting details. These strategies include things like omitting servers with only one document on them, and collapsing servers with several documents

- rendering information as properties of graphical entities (e.g. colours, thicknesses etc.).

This approach is taken to ensure that, as far as possible, rules can be written at the domain level, without reference to the mechanics of fetching or searching documents. The figures give some idea of the extent to which this has been achieved.

Figure 1 shows the rule that is responsible for deciding to fetch a page. This is perfectly ordinary CLIPS rule matching

```
(defrule decide-to-fetch-page
    "If we know of a page relevant to our target,
        note that we should fetch it"
    (declare (salience -60))
    (target ?query ?function)
    (object (is-a page)
    (name ?doc)
    (type html|unknown)
    (did ?*not-fetched*)
    (url ?url&:(perl-test ?function ?url)))
=>
    (assert (to-fetch ?query 1 ?url ?doc)))
```

Figure 1: A typical rule, written in domain terms

against a fact and an object, and using a user-defined function to test the value of one of the object's slots (url). The only unusual feature is the fact that the actual test on the URL is carried out in Perl. The simplest way to have done this would have been to directly match the URL against a Perl regular expression defining the pages of interest by using the CAPE function p-match-p, thus

(url ?url\&:(p-match-p ?url ?regexp))

In fact, though, the rule used is somewhat more sophisticated in its use of Perl facilities. At the time when the target for the query is defined, the regular expression is passed to a Perl function which defines a second Perl function to check a URL against the regular expression, and returns the name of that function. Because this function is only used to match against a single (constant) regular expression, Perl can be told to optimise the matching process. The specially-defined function is then called using the CAPE function perl-test, which calls the Perl function named by first argument, and returns a CLIPS boolean. Other arguments to perl-test are converted to Perl primitives (in this case, the URL is converted to a Perl string) and passed in to the Perl function.

Figure 2 illustrates a second rule, which recognises a group of documents that are "siblings"—i.e. that contain links to the same URL which have the same anchor text. The rule matches over facts describing the links that have been found in the various documents, finding three that refer to different document identifiers (?doc-id, ?doc-id2 and ?doc-id3 but are otherwise identical. Facts like these can be generated in CAPE in a single line of code, just by requesting the matches for an appropriate regular expression. They could, of course, be generated in CLIPS, but it would require considerably more error-prone and tedious programming.

Note also the use of a second Perl function (same-doc) which compares two URLs to determine whether they refer to the same document. This involves more than just checking whether the URLs are identical, since it is also necessary to ignore any anchors within the document, and to disregard any trailing slash.

6. STATUS AND FUTURE WORK

CAPE is a new tool which brings together two very different programming systems. It runs under a number of versions of UNIX, including Solaris and Linux. The core functionality of the current version of the system has been stable since the spring of 1998, and, as described above, the system itself has been used for the development of a number of research systems since the following summer.

37

```
(defrule spot-siblings
    "Find 'identical' links to the same place from
     different sources"
    (declare (salience 50))
    (link ?doc-id ?source ?anchor ?dest $?rest)
    (not
        (object (is-a sibling-set)
                (anchor ?anchor)
                (destination ?dest)))
    (link ?doc-id2&~?doc-id
          ~?source ?anchor ?dest $?rest)
    (link ?doc-id3&~?doc-id&~?dod-id2
          ~?source ?anchor ?dest $?rest)
    (object (is-a webitem)
            (name ?obj)
            (url ?u&:(perl-test same_doc ?dest ?u)))
    =>
    (bind ?ss (gensym "family-"))
    (make-instance ?ss of sibling-set
                   (intarget TRUE)
                   (status possible)
                   (anchor ?anchor)
                   (destination-obj ?obj)
                   (destination ?dest)))
```

Figure 2: A second rule, again almost entirely in domain terms

The system was made available by FTP and announced on the Web in February 1999, and has been being downloaded over one hundred and fifty times since then.

The core CAPE environment comprises 3000 lines of C code, plus 600 lines of Perl (and a small amount of CLIPS). The system comes with a thirty-one page manual ([12]) and 1500 lines of CAPE code in two standard CAPE "component applications", or "Capplets" (support for regression testing CAPE applications and a simple Web server).

The distribution currently also includes three demonstration applications:

Handshaking: A minimal demonstration of communication between a pair of CAPE processes. An "interface agent" accepts user requests (via HTTP) for commands to be executed, and forwards them to the relevant "execution agent", which executes them in due course. The execution agent then contacts the interface agent with the results, which it then forwards to the user.

Web server This application operates as a (partial) web server. It accepts a limited set of HTTP requests, locates the relevant file and then replies with either an appropriate HTTP response—either an error, or the contents of the requested file (along with appropriate HTTP header information). However, the server also allows users to specify ((using an HTML form that it generates) required transformations to pages, which are then subsequently applied to the pages being served. This demonstration system, which uses the "webserve" component application, is about 350 lines of code.

"Dungeon" This application provides a real-time multi-user "dungeon" game. Players access the system via a web browser, and are able to direct the actions of one of a number of "characters" moving within a simple environment. In addition to moving about and

manipulating objects, players are likely to encounter other characters, either controlled by other players, or by the computer. All descriptions of situations are generated from information structures (i.e. there is no "canned" text) based on the objects and locations known to the user. Computer-controlled characters can generate and follow multi-step plans.

The entire application was build from scratch in five days. It contains a total of 2800 lines of CAPE code (about one third Perl), although re-implementation using the webserve component application would reduce this substantially.

One of the main difficult features of CAPE is the fact that CLIPS and Perl have different syntaxes, CLIPS being Lisp-like and Perl being generally (arguably "vaguely") C-like. This unfortunately means that fully exploiting CAPE requires reasonable proficiency in both languages, *and* in working with more than one language at a time. The possibility of producing a single "unified" syntax has considerable appeal. However, defining such a thing is not easy, since Perl syntax is already complex and very "dense'—that is, a high proportion of characters and character combinations are already assigned meanings. Moreover, the appeal of a single syntax must be seen alongside the advantages of continuing to support unchanged the syntax of each of the underlying languages: access to the substantial bodies of software, documentation and expertise for these languages.

There are a number of obvious extensions to CAPE, some of which will be added to the system in the near future:

- Use the code pre-processing facilities extend CLIPS to support backward chaining, or goal-oriented reasoning, by using particular fact patterns to represent the presence of particular goals, which will then drive the forward-chaining rule-engine in CLIPS.

- Use the code pre-processing facilities extend CLIPS to support explicit declaration of relationship properties (e.g. transitivity, reflexivity, symmetry etc.) and the automatic definition of rules and methods to enforce and infer the consequences of this.

- CLIPS and Perl both provide powerful object systems (albeit with quite different properties) and support for modules (multiple namespaces). They should be integrated.

- CLIPS has powerful mechanisms for directing I/O by defining (in C) arbitrary *routers* for various logical information types. Perl has sophisticated mechanisms for formatting and paginating output. Yet CAPE's own mechanisms for handling I/O through sockets is not yet related to either!

- There are several additions to CAPE which would help implementing autonomous agents: specifically, mechanisms to allow the system to do things at a particular time, to facilitate the controlled mapping of specific kinds of information between CLIPS working memory and (ideally shared) persistent store and to support the KQML agent-communication standard.

- Recent versions of Perl can be compiled to support multiple threads. In contrast, CLIPS is only single-threaded, and although CAPE can be *linked* with a

multi-threaded Perl, it does nothing to allow multiple threads to be actually used.

7. CONCLUSION

CAPE is a powerful tool for building a new generation of KBSs. It combines the strengths of two well-established tools with very powerful but complementary pattern-matching mechanisms. Perl's ability to search text and match powerful regular expressions is unequaled, while CLIPS provides powerful mechanisms for finding patterns of combinations of symbolic information. The CAPE programmer can exploit the strengths of both, using Perl to analyse documents or query results, and CLIPS to recognise and react to the combinations of matches found.

CAPE provides powerful mechanisms to support a number of key activities:

Symbolic reasoning CLIPS offers a very efficient forward chaining rule-based system with extremely expressive pattern matching, coupled with a highly flexible object-oriented system and supported by a truth-maintenance system.

Data analysis/manipulation Perl has extremely powerful regular expression matching coupled with very concise string handling and easy-to-use hash-based index-building and data structuring.

Service Provision CAPE's socket monitoring mechanisms allow a rule-based program to remain responsive to external activity even while it is reasoning.

Standard languages/libraries CAPE programs can use any of the enormous range of software available in CPAN, the Comprehensive Perl Archive Network (accessible via http://www.perl.com/).

Interaction with software packages Perl provides very concise and flexible mechanisms for controlling and processing the results obtained from system commands and other external programs. CAPE programmers can also exploit the tools for generating Perl "wrappers" for software components written in C, and make use of Perl's ability to dynamically load compiled code at run-time.

Together, these features make CAPE a powerful tool for building KBSs that can exploit the opportunities offered by the Internet.

CAPE is freely available, with full source, from
> http://www.hcrc.ed.ac.uk/~robert/CAPE

ACKNOWLEDGEMENTS

REFERENCES

[1] A. Sloman and S. Hardy. "Poplog: A Multi-Purpose Multi-Language Program Development Environment" AISB Quarterly, Vol 47, pp26-34, 1983.

[2] Gary Riley, "Clips: An Expert System Building Tool". Proceedings of the Techlology 2001 Conference, San Jose, CA, 1991

[3] L. Wall, T. Christiansen and R. Schwartz, *Programming Perl (2nd Edn.)* O'Reilly and Associates, 1996.

[4] J. Giarratano and G. Riley, "Expert Systems: Principles and Programming, 2nd Edition". PWS Publishing Company, 1994

[5] S. Srinivasan, "Advanced Perl Programming". O'Reilly and Associates, 1997.

[6] R. Inder, "State of the ART: A review of the Automated Reasoning Tool." In *Expert System Applications*, S. Vadera (ed), Sigma Press, Wilmslow, 1989. Also available as AIAI-TR-41.

[7] Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence 19(1982), 17-37.

[8] Robert Inder, Nadia Bianchi and Toshikazu Kato, "K-DIME: A Software Framework for Kansei Filtering of Internet Material" Proceedings of IEEE Systems Man and Cybernetics, 1999.

[9] Robert Inder and Toshikazu Kato, "Towards Shoppers' Assistants: Agents to Combine Information". In *Advanced Database Systems for Integration of Media and User Environments 98*, Y. Kambayashi (ed.), World Scientific, February 1998.

[10] Robert Inder, Matthew Hurst and Toshkazu Kato, "A Prototype Agent to Assist Shoppers" in *Computer Networks and ISDN Systems*, V30 (1998), pp 643-645. Full version available as *Technical Report ETL-TR98-3*, Electrotechnical Laboratory, Tsukuba, Japan.

[11] Kozo Sugiyama and Kazuo Misue, "Visualization of Structural information: Automatic Drawing of Compound Digraphs,". IEEE Transactions on Systems, Man, and Cybernetics, vol. 21. 1991.

[12] Robert Inder. CAPE *Users Manual.* Technical Report ETL-TR98-3,[1] Electrotechnical Laboratory, Tsukuba, Japan.

---

[1] The most recent version is always available via http://www.hcrc.ed.ac.uk/~robert/CAPE/manual.ps